20 | RDD和DataFrame：既生瑜、何生亮

2021-04-28 吴磊

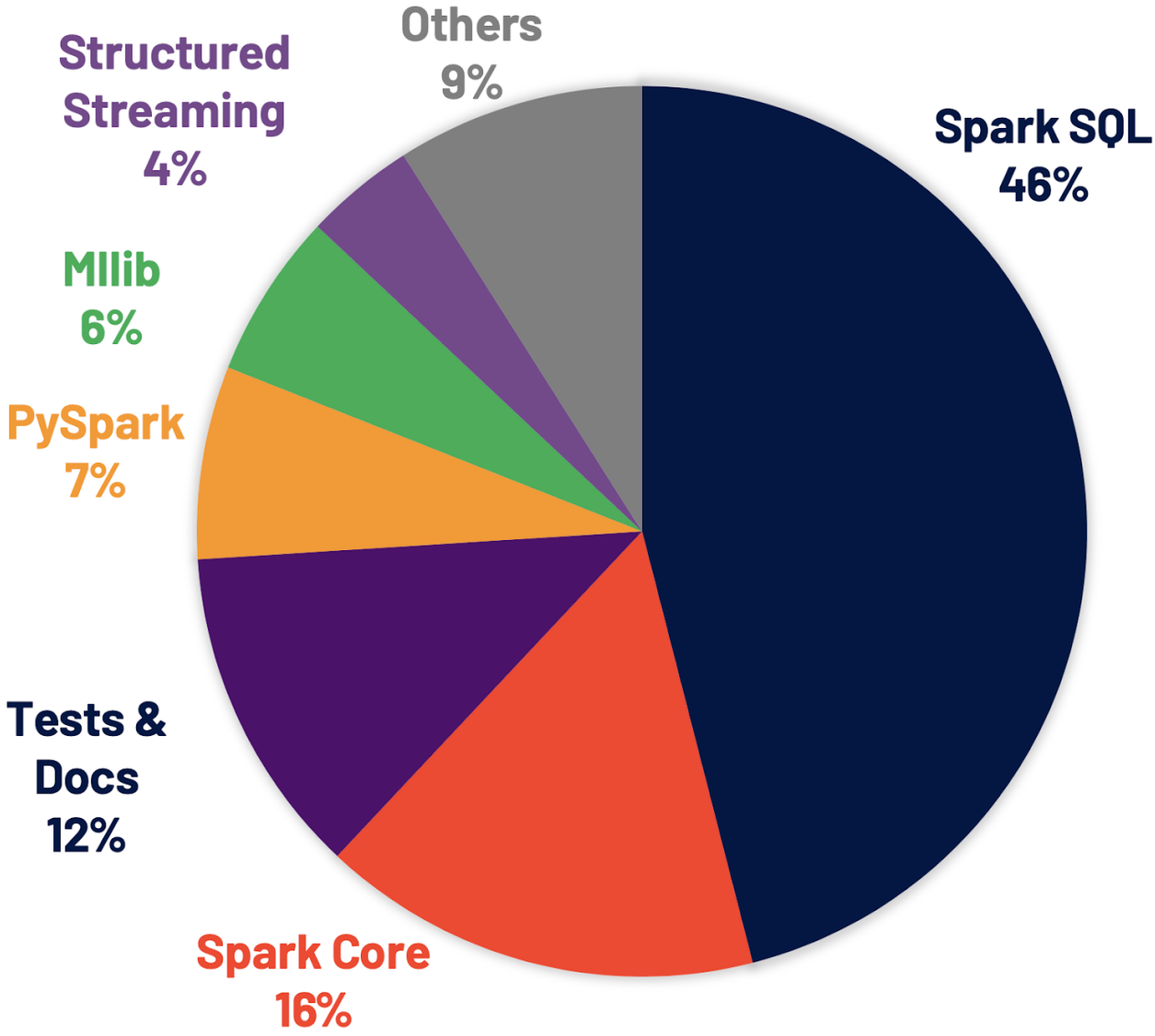
Spark性能调优实战 进入课程 >

讲述：吴磊
时长 15:38 大小 14.33M

你好，我是吴磊。

从今天开始，我们进入 Spark SQL 性能调优篇的学习。在这一篇中，我会先带你学习 Spark SQL 已有的优化机制，如 Catalyst、Tungsten 这些核心组件，以及 AQE、DPP 等新特性。深入理解这些内置的优化机制，会让你在开发应用之初就有一个比较高的起点。然后，针对数据分析中的典型场景，如数据关联，我们再去深入探讨性能调优的方法和技巧。

今天这一讲，我们先来说说 RDD 和 DataFrame 的渊源。这也是面试的时候，面试官会问的。比如说：“Spark 3.0 大版本发布，Spark SQL 的优化占比将近 50%；而像 PySpark、Mllib 和 Streaming 的优化占比都不超过 10%，Graph 的占比几乎可以忽略不计。这是否意味着 Spark 社区逐渐放弃了其他计算领域，只专注于数据分析？”



这个问题的标准答案是：“Spark SQL 取代 Spark Core，成为新一代的引擎内核，所有其他子框架如 Mllib、Streaming 和 Graph，都可以共享 Spark SQL 的性能优化，都能从 Spark 社区对于 Spark SQL 的投入中受益。”不过，面试官可没有那么好对付，一旦你这么说，他 / 她可能会追问：“为什么需要 Spark SQL 这个新一代引擎内核？Spark Core 有什么问题吗？Spark SQL 解决了 Spark Core 的哪些问题？怎么解决的？”

面对这一连串“箭如雨发”的追问，你还能回答出来吗？接下来，我就从 RDD 的痛点说起，一步一步带你探讨 DataFrame 出现的必然性，Spark Core 的局限性，以及它和 Spark SQL 的关系。

RDD 之痛：优化空间受限

自从 Spark 社区在 1.3 版本发布了 DataFrame，它就开始代替 RDD，逐渐成为开发者的首选。我们知道，新抽象的诞生一定是为了解决老抽象不能搞定的问题。那么，这些问题都是什么呢？下面，我们就一起来分析一下。

在 RDD 的开发框架下，我们调用 RDD 算子进行适当的排列组合，就可以很轻松地实现业务逻辑。我把这些使用频繁的 RDD 算子总结到了下面的表格里，你可以看一看。

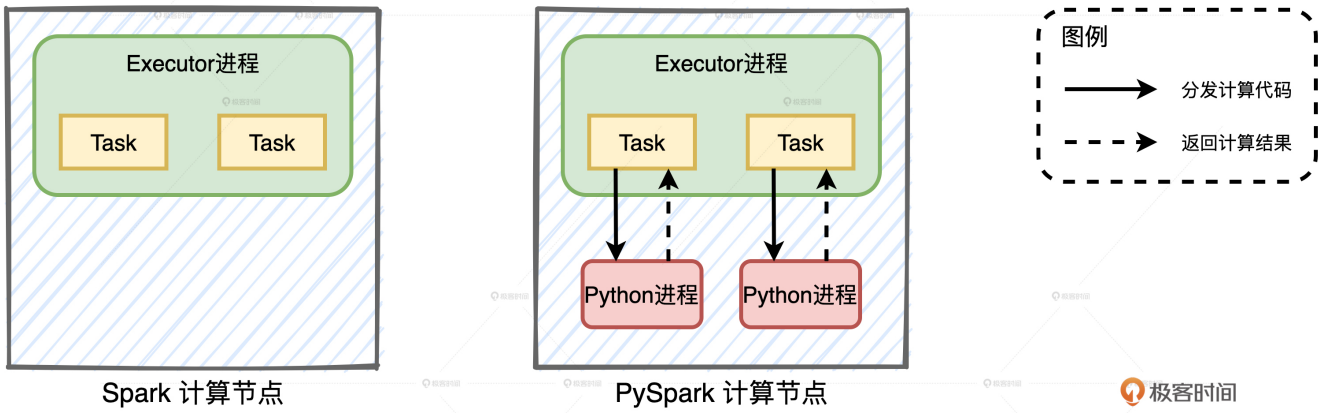
算子类型	适用范围	算子用途	算子集合		
Transformations	任意RDD	单个RDD 数据转换	map mapPartitions mapPartitionsWithIndex	filter	flatMap
	任意RDD	与其他RDD 进行组合	union intersection	join cogroup cartesian	
	任意RDD	数据采样	sample	distinct	
	任意RDD	数据分片 的重分布	coalesce	repartition repartitionAndSortWithinPartitions	
	Paired RDD	单个RDD 数据聚合	groupByKey sortByKey	reduceByKey aggregateByKey	
Actions	任意RDD	收集数据到Spark Driver	collect first take	takeSample takeOrdered	count
	任意RDD	数据持久化 到存储系统	saveAsTextFile saveAsSequenceFile saveAsObjectFile		
	任意RDD	函数式编程 副作用操作	foreach	极客时间	

表格中高亮显示的就是 RDD 转换和聚合算子，它们都是高阶函数。高阶函数指的是形参包含函数的函数，或是返回结果包含函数的函数。为了叙述方便，我们把那些本身是高阶函数的 RDD 算子，简称“高阶算子”。

对于这些高阶算子，开发者需要以 Lambda 函数的形式自行提供具体的计算逻辑。以 map 为例，我们需要明确对哪些字段做映射，以什么规则映射。再以 filter 为例，我们需要指明以什么条件在哪些字段上过滤。

但这样一来，Spark 只知道开发者要做 map、filter，但并不知道开发者打算怎么做 map 和 filter。也就是说，**在 RDD 的开发模式下，Spark Core 只知道“做什么”，而不知道“怎么做”。**这会让 Spark Core 两眼一抹黑，除了把 Lambda 函数用闭包的形式打发到 Executors 以外，实在是没有什么额外的优化空间。

对于 Spark Core 来说，优化空间受限最主要的影响，莫过于让应用的执行性能变得低下。一个典型的例子，就是相比 Java 或者 Scala，PySpark 实现的应用在执行性能上相差悬殊。原因在于，在 RDD 的开发模式下，即便是同一个应用，不同语言实现的版本在运行时也会有着天壤之别。



不同语言的运行时计算过程

当我们使用 Java 或者 Scala 语言做开发时，所有的计算都在 JVM 进程内完成，如图中左侧的 Spark 计算节点所示。

而当我们在 PySpark 上做开发的时候，只能把由 RDD 算子构成的计算代码，一股脑地发送给 Python 进程。Python 进程负责执行具体的脚本代码，完成计算之后，再把结果返回给 Executor 进程。由于每一个 Task 都需要一个 Python 进程，如果 RDD 的并行度为 #N，那么整个集群就需要 #N 个这样的 Python 进程与 Executors 交互。不难发现，其中的任务调度、数据计算和数据通信等开销，正是 PySpark 性能低下的罪魁祸首。

DataFrame 应运而生

针对优化空间受限这个核心问题，Spark 社区痛定思痛，在 2013 年在 1.3 版本中发布了 DataFrame。那么，DataFrame 的特点是什么，它和 RDD 又有什么不同呢？

首先，用一句话来概括，DataFrame 就是携带数据模式（Data Schema）的结构化分布式数据集，而 RDD 是不带 Schema 的分布式数据集。**因此，从数据表示（Data Representation）的角度来看，是否携带 Schema 是它们唯一的区别。**带 Schema 的数据表示形式决定了 DataFrame 只能封装结构化数据，而 RDD 则没有这个限制，所以除了结构化数据，它还可以封装半结构化和非结构化数据。

其次，从开发 API 上看，**RDD 算子多是高阶函数，这些算子允许开发者灵活地实现业务逻辑，表达能力极强。**

DataFrame 的表达能力却很弱。一来，它定义了一套 DSL（Domain Specific Language）算子，如 select、filter、agg、groupBy 等等。由于 DSL 语言是为解决某一类任务而专门设计的计算机语言，非图灵完备，因此，语言表达能力非常有限。二来，DataFrame 中的绝大多数算子都是标量函数（Scalar Functions），它们的形参往往是结构化的数据列（Columns），表达能力也很弱。

你可能会问：“相比 RDD，DataFrame 的表示和表达能力都变弱了，那它是怎么解决 RDD 优化空间受限的核心痛点呢？”

当然，仅凭 DataFrame 在 API 上的改动就想解决 RDD 的核心痛点，比登天还难。**DataFrame API 最大的意义在于，它为 Spark 引擎的内核优化打开了全新的空间。**

首先，DataFrame 中 Schema 所携带的类型信息，让 Spark 可以根据明确的字段类型设计定制化的数据结构，从而大幅提升数据的存储和访问效率。其次，DataFrame 中标量算子确定的计算逻辑，让 Spark 可以基于启发式的规则和策略，甚至是动态的运行时信息去优化 DataFrame 的计算过程。

Spark SQL 智能大脑

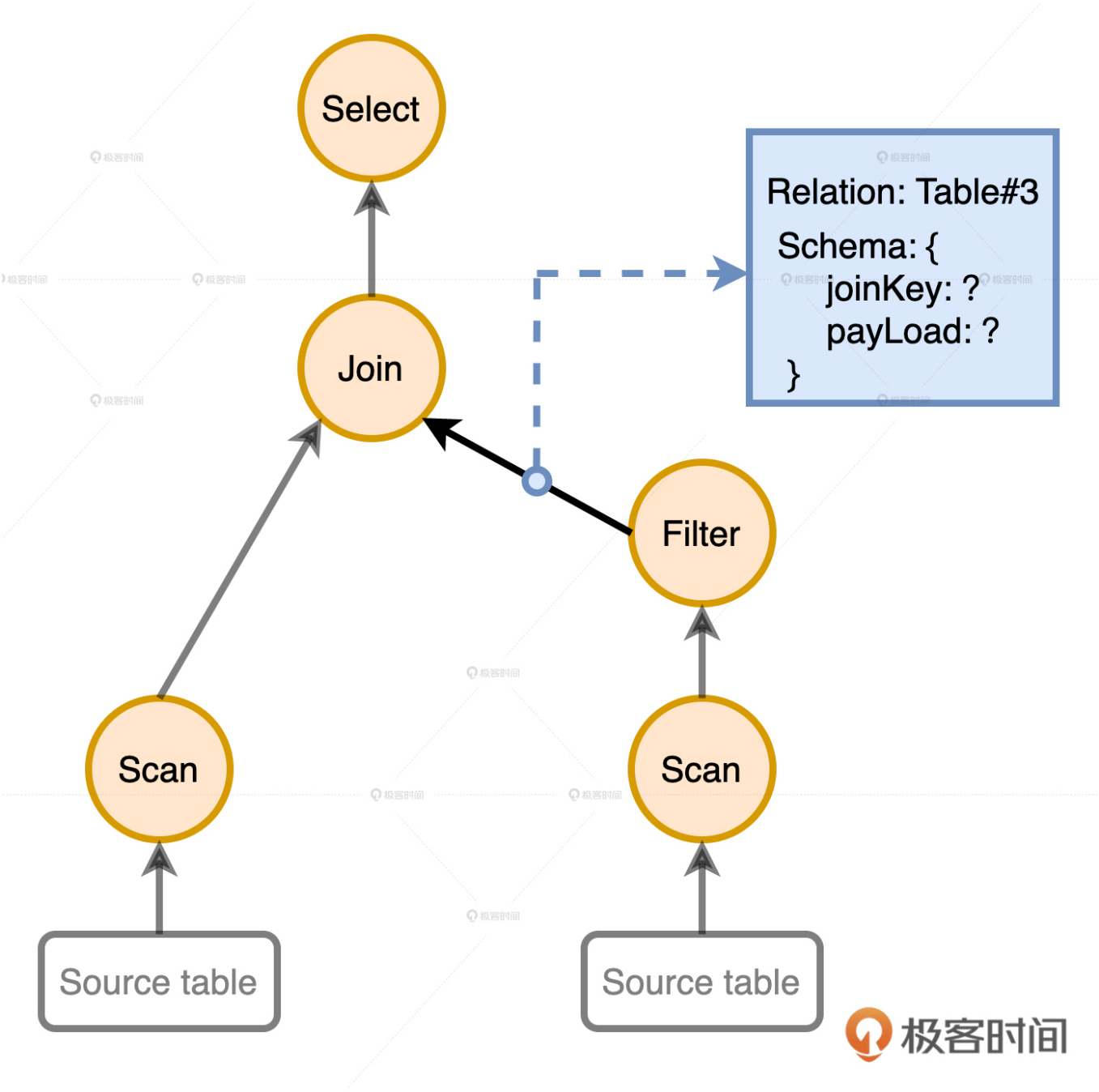
那么问题来了，有了 DataFrame API，负责引擎内核优化的那个幕后英雄是谁？为了支持 DataFrame 开发模式，Spark 从 1.3 版本开始推出 Spark SQL。**Spark SQL 的核心组件有二，其一是 Catalyst 优化器，其二是 Tungsten。**关于 Catalyst 和 Tungsten 的特性和优化过程，我们在后面的两讲再去展开，今天这一讲，咱们专注在它们和 DataFrame 的关系。

Catalyst：执行过程优化

我们先来说说 Catalyst 的优化过程。当开发者通过 Actions 算子触发 DataFrame 的计算请求时，Spark 内部会发生一系列有趣的事情。

首先，基于 DataFrame 确切的计算逻辑，Spark 会使用第三方的 SQL 解析器 ANTLR 来生成抽象语法树（AST，Abstract Syntax Tree）。既然是树，就会有节点和边这两个基本

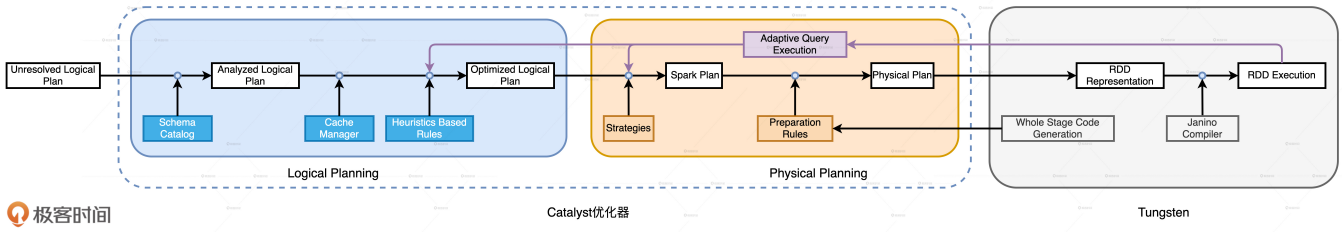
的构成元素。节点记录的是标量算子（如 select、filter）的处理逻辑，边携带的是数据信息：关系表和数据列，如下图所示。这样的语法树描述了从源数据到 DataFrame 结果数据的转换过程。



AST语法树示意图

在 Spark 中，语法树还有个别名叫做“Unresolved Logical Plan”。它正是 Catalyst 优化过程的起点。之所以取名“Unresolved”，是因为边上记录的关系表和数据列仅仅是一些字符串，还没有和实际数据对应起来。举个例子，Filter 之后的那条边，输出的数据列是 joinKey 和 payLoad。这些字符串的来源是 DataFrame 的 DSL 查询，Catalyst 并不确定这些字段名是不是有效的，更不知道每个字段都是什么类型。

因此，**Catalyst 做的第一步优化，就是结合 DataFrame 的 Schema 信息，确认计划中的表名、字段名、字段类型与实际数据是否一致**。这个过程也叫做把“Unresolved Logical Plan”转换成“Analyzed Logical Plan”。



Spark SQL的端到端优化过程

基于解析过后的“Analyzed Logical Plan”，Catalyst 才能继续做优化。利用启发式的规则和执行策略，Catalyst 最终把逻辑计划转换为可执行的物理计划。总之，Catalyst 的优化空间来源 DataFrame 的开发模式。

Tungsten：数据结构优化

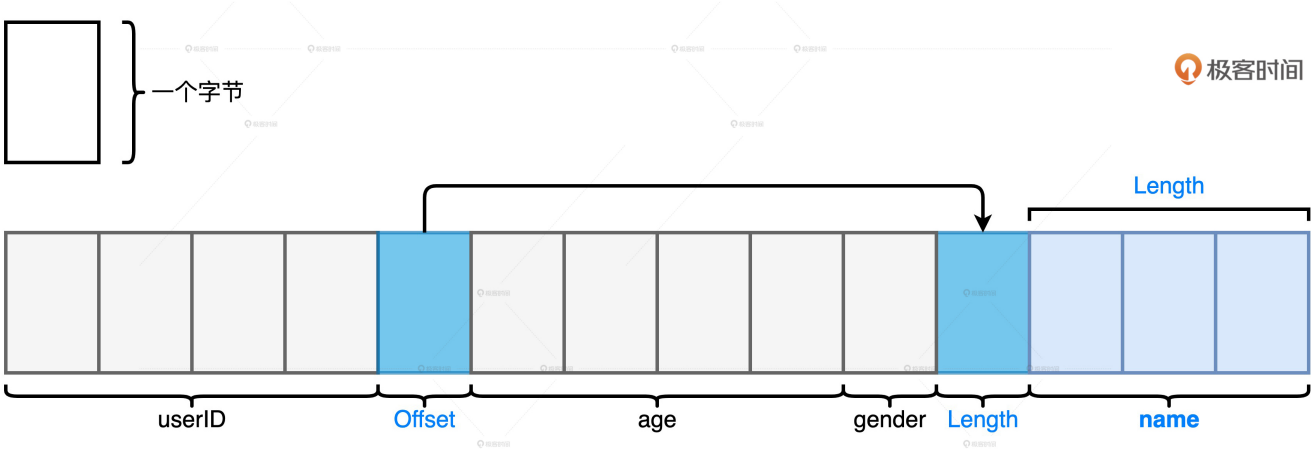
说完 Catalyst，我接着再来说说 Tungsten。在开发原则那一讲，我们提到过 Tungsten 使用定制化的数据结构 Unsafe Row 来存储数据，Unsafe Row 的优点是存储效率高、GC 效率高。Tungsten 之所以能够设计这样的数据结构，仰仗的也是 DataFrame 携带的 Schema。Unsafe Row 我们之前讲过，这里我再带你简单回顾一下。

字段	数据类型	含义
userID	Int	用户ID
name	String	姓名
age	Int	年龄
gender	Char	性别

极客时间

结构化的二维表

Tungsten 是用二进制字节序列来存储每一条用户数据的，因此在存储效率上完胜 Java Object。比如说，如果我们要存储上表中的数据，用 Java Object 来存储会消耗 100 个字节数，而使用 Tungsten 仅需要不到 20 个字节，如下图所示。

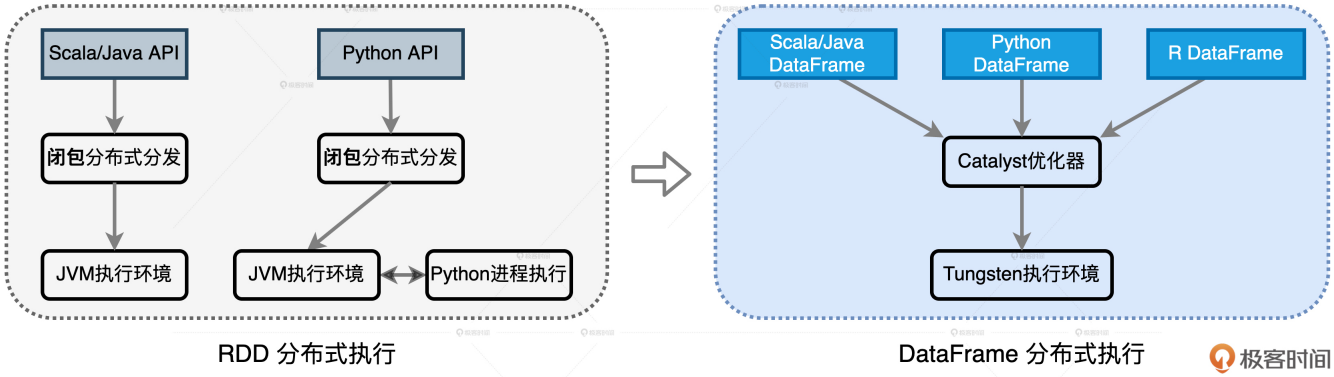


二进制字节序列

但是，要想实现上图中的二进制序列，Tungsten 必须要知道数据条目的 Schema 才行。也就是说，它需要知道每一个字段的数据类型，才能决定在什么位置安放定长字段、安插 Offset，以及存放变长字段的数据值。DataFrame 刚好能满足这个前提条件。

我们不妨想象一下，如果数据是用 RDD 封装的，Tungsten 还有可能做到这一点吗？当然不可能。这是因为，虽然 RDD 也带类型，如 RDD[Int]、RDD[(Int, String)]，但如果 RDD 中携带的是开发者自定义的数据类型，如 RDD[User]或是 RDD[Product]，Tungsten 就会两眼一抹黑，完全不知道你的 User 和 Product 抽象到底是什么。成也萧何、败也萧何，RDD 的通用性是一柄双刃剑，在提供开发灵活性的同时，也让引擎内核的优化变得无比困难。

总的来说，基于 DataFrame 简单的标量算子和明确的 Schema 定义，借助 Catalyst 优化器和 Tungsten，Spark SQL 有能力在运行时构建起一套端到端的优化机制。这套机制运用启发式的规则与策略，以及运行时的执行信息，将原本次优、甚至是低效的查询计划转换为高效的执行计划，从而提升端到端的执行性能。因此，在 DataFrame 的开发框架下，不论你使用哪种开发语言，开发者都能共享 Spark SQL 带来的性能福利。



不同开发模式下的分布式执行过程

最后，我们再来回顾最开始提到的面试题：“从 2.0 版本至今，Spark 对于其他子框架的完善与优化，相比 Spark SQL 占比很低。这是否意味着，Spark 未来的发展重心是数据分析，其他场景如机器学习、流计算会逐渐边缘化吗？”

最初，Spark SQL 确实仅仅是运行 SQL 和 DataFrame 应用的子框架，但随着优化机制的日趋完善，Spark SQL 逐渐取代 Spark Core，演进为新一代的引擎内核。到目前为止，所有子框架的源码实现都已从 RDD 切换到 DataFrame。因此，和 PySpark 一样，像 Streaming、Graph、Mllib 这些子框架实际上都是通过 DataFrame API 运行在 Spark SQL 之上，它们自然可以共享 Spark SQL 引入的种种优化机制。

形象地说，Spark SQL 就像是 Spark 的智能大脑，凡是通过 DataFrame 这双“眼睛”看到的问题，都会经由智能大脑这个指挥中心，统筹地进行分析与优化，优化得到的行动指令，最终再交由 Executors 这些“四肢”去执行。

小结

今天，我们围绕 RDD 的核心痛点，探讨了 DataFrame 出现的必然性，Spark Core 的局限性，以及它和 Spark SQL 的关系，对 Spark SQL 有了更深刻的理解。

RDD 的核心痛点是优化空间有限，它指的是 RDD 高阶算子中封装的函数对于 Spark 来说完全透明，因此 Spark 对于计算逻辑的优化无从下手。

相比 RDD，DataFrame 是携带 Schema 的分布式数据集，只能封装结构化数据。DataFrame 的算子大多数都是普通的标量函数，以消费数据列为主。但是，DataFrame 更弱的表示能力和表达能力，反而为 Spark 引擎的内核优化打开了全新的空间。

根据 DataFrame 简单的标量算子和明确的 Schema 定义，借助 Catalyst 优化器和 Tungsten，Spark SQL 有能力在运行时，构建起一套端到端的优化机制。这套机制运用启发式的规则与策略和运行时的执行信息，将原本次优、甚至是低效的查询计划转换为高效的执行计划，从而提升端到端的执行性能。

在 DataFrame 的开发模式下，所有子框架、以及 PySpark，都运行在 Spark SQL 之上，都可以共享 Spark SQL 提供的种种优化机制，这也是为什么 Spark 历次发布新版本、Spark SQL 占比最大的根本原因。

每日一练

1. Java Object 在对象存储上为什么会有比较大的开销？JVM 需要多少个字节才能存下字符串“abcd”？
2. 在 DataFrame 的开发框架下， PySpark 中还有哪些操作是“顽固分子”，会导致计算与数据在 JVM 进程与 Python 进程之间频繁交互？(提示：参考 RDD 的局限性，那些对 Spark 透明的计算逻辑，Spark 是没有优化空间的)

期待在留言区看到你的思考和答案，我们下一讲见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 网络视角：如何有效降低网络开销?

下一篇 21 | Catalyst逻辑计划：你的SQL语句是怎么被优化的？（上）

精选留言 (8)

写留言

sky_sql
2021-04-29

老师好，DataFrame和RDD这两套独立的API，最终还是会转化为RDD之上的计算吧?

展开

作者回复: 对，没错，DataFrame的code，经过Spark SQL优化之后，最终交由Tungsten生成代码和RDD[InternalRow]，代码交由DAGScheduler进行分发、运行在RDD[InternalRow]之上。

后面的21、22、23会详细展开从DataFrame到Catalyst、再到Tungsten，最终生成代码和RDD的过程，可以重点关注下哈~

2

Geek_d794f8
2021-04-29

磊哥，Tungsten 使用定制化的数据结构 Unsafe Row 来存储数据，是不是指的cache缓存的时候使用Unsafe Row存储的数据，或是shuffle溢写到磁盘的时候存储的结构，还是说从数据源读进来的时候会转化成Unsafe Row的结构?

展开

作者回复: 都对，或者换句话说，只要是走Spark SQL的流程（DataFrame、Dataset、SQL三种API），就都能利用到Tungsten的这种二进制Unsafe Row。比如，就像你说的：

1）缓存的时候，可以利用得到，Tungsten用Unsafe Row来封装每条数据记录；
2）数据源读进来的时候（当然这么说不太严谨，严格来说，应该是Tungsten生成的“手写代码”交付执行的时候，在运行时执行读取数据源的时候，会把它convert成Tungsten的二进制Unsafe Row）；
3）Shuffle的时候也是一样，不仅如此，Shuffle还能利用文中介绍的Memory Page（一个JVM对象），来进一步提升内存的利用效率。

1

sky_sql
2021-04-28

RDD是Spark对于分布式数据模型的抽象，调度、存储都是RDD维度的，DataFrame底层是使用RDD的算子实现的吧?
对于普通开发者后面使用DataFrame实现业务逻辑，尽量不使用RDD?

展开

作者回复: DataFrame底层不是RDD实现的，DataFrame和RDD是两套独立的API。区别在于，RDD的优化引擎是Spark Core；而DataFrame的优化引擎是Spark SQL，这个我们后面3讲：21、22、23会详细的介绍哈~

对于开发者来说，强烈推荐DataFrame开发API，放弃RDD开发API。Spark SQL的优化机制远胜于Spark Core，换句话说，同样的业务应用，仅仅是API不同，你用DataFrame开发的code，天然地就比你用RDD开发的code，在执行效率上面更高，啥都不用调优，天然就更快。

1 1

aof
2021-05-05

PySpark应用中如果用到了Python里面的数据结构都会导致在JVM进程和Python进程间进行交互的吧

展开

作者回复: 对没错，除此之外，还有万恶的UDF。

1

Z字锤锤
2021-05-02

Java Object 在对象存储上为什么会有比较大的开销?
我查阅了了一些资料，Java对象的存储在JVM上至少需要32或64Bit的字节存储对象自身信息，哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳。
"ABCD"的存储至少需要8个字节。

展开

作者回复: 是的，你说的这些开销都对，不过“ABCD”在JVM中实际需要48个字节才能存下哈~我怀疑你是不是漏打了个4，哈哈~

具体计算细节可以参考这里：https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

1

Z字锤锤
2021-05-02

RDD和DataFrame的最大区别，DF像是一张二维表，可以用来写SQL。RDD只能使用算子。

展开

作者回复: DF确实就是二维表，和RDD相比，除了API的区别之外，最大的区别，还是DF可以走Spark SQL做优化~

1 1

Fendora范东_
2021-04-30

对RDD和DataFrame的理解
1.首先区分这两个东西，RDD分布式数据集;DF是带数据模式的结构化分布式数据集。核心区别在于DF带数据模式(感觉像传统分布式数据库中的一张表)，RDD不带数据模式或者说

是泛型的。
2.其次是这两者分别提供的API，RDD API优化引擎是Spark Core(没理解这句话，我理...
展开

作者回复: Perfect x 4！ 理解得非常到位了~

老弟的概括能力很强，确实，把握这4点，关于RDD和DataFrame的区别与联系，其实就算是吃透了。

关于你说的第二点，确实，Spark Core实际就是分布式运行时，负责分布式任务调度、执行，严格来说，并不存在什么优化机制，仅仅是分发分布式代码、运行、状态交互，等等。

第四点说得尤其到位，大赞👍

Geek_d794f8
2021-04-29

Tungsten的这种存储数据结构，感觉和parquet类似，他们之间有什么关联吗
展开

作者回复: 确实有相似之处，毕竟都是用二进制的形式来存储数据，所以会有相通的地方。

不过，两者没啥联系哈，实际上，我倒是觉得，Parquet的数据结构，要比Tungsten的复杂得多。